A DESIGN APPROACH FOR DYNAMIC RECONFIGURATION OF UNATTENDED SENSORS, UNMANNED SYSTEMS, AND MONITORING STATIONS

Matthew W. Skalny and William Smuda
TARDEC Robotics Mobility Lab
Warren, MI 48397-5000

ABSTRACT

The design and implementation of software for networked systems of diverse physical assets is a continuing challenge to robotic and network sensor developers. The problems are often multiplied when reconfiguring or adding new elements to existing designs in order to meet the demands of changing tactics and missions, and to meet new requirements for interoperability and additional capabilities. Systems are often designed in a way such that configuration and reconfiguration may be difficult, time consuming, and costly. Interoperability between new and legacy systems may require significant changes to the code and design of a system. Static or closed designs can lead to a system that is difficult to add new sensors or payloads to. In this paper, we describe a design approach that utilizes **Model Driven Engineering (MDE)**, the **OSGi Service Platform framework (OSGi)**, and an **open, flexible services oriented architecture** to **maximize software reuse** and to **ensure rapid development of new features and capabilities to meet the changing requirements for unmanned systems**.

1. INTRODUCTION

1.1 Model Driven Engineering

Model Driven Engineering focuses on abstractions particular to the application problem space and expresses designs in terms of concepts from that space (Schmidt, 2006). MDE combines software components to conform to specific design patterns with Domain Specific Languages. These languages describe a Meta Model, often graphical, that defines the relationships of abstractions in the domain. Domain engineers then create concrete instances of the Meta Model using icons that represent available services and components for the composition of the final design. Using this completed design, program generators assemble the services and components and create the glue code that allows them to work together as a single, cohesive system.

Figure 1 shows a Meta Model for a robot system done using the open source Generic Modeling Environment (GME). The model is done using the Unified Modeling Language (UML). In this case, the top level model element is labeled "Robot". The robot Meta Model contains both messages and artifacts – abstract services with no implementation defined. Five different atoms provide possible implementations of artifacts. Artifacts can send or receive zero or more different messages.

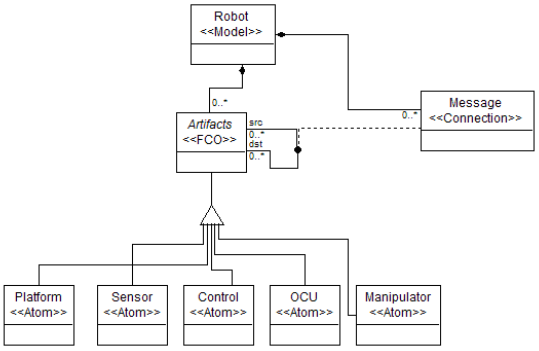

Figure 1: Simple Meta Model for a Robot

The Meta Model shown in figure 1 may be used to create a domain specific composition as shown in figure 2. The model in figure 2 has three robots, a leader and two followers. Each robot has a GPS positioning sensor and the two followers have distance sensors. The waypoint driver control computer computes waypoints for the two followers based on the input from the five sensors, and passes new messages to the primitive driver to control the two follower robots.

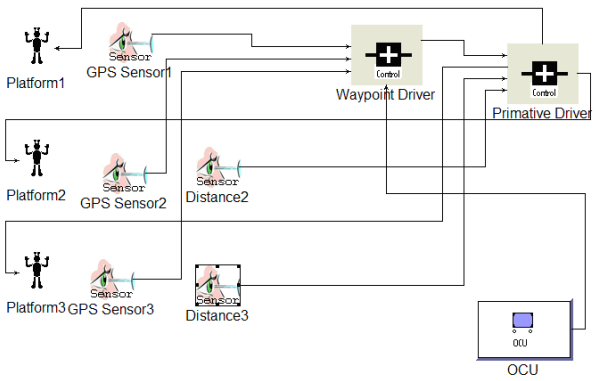

Figure 2: Domain Model for Multiple Robot System

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 NOV 2006	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Design Approach For Dynamic Reconfiguration Of Unattended Sensors, Unmanned Systems, And Monitoring Stations	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **TARDEC Robotics Mobility Lab Warren, MI 48397-5000**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM002075., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**UU**	**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1.2. OSGi and Services Oriented Architecture

From the OSGi alliance website (OSGi Alliance, 2006), "OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java™ platform." A basic setup using OSGi involves four main levels – the hardware, the Java Virtual Machine (JVM), the OSGi framework, and the application layer. The hardware is essentially any device capable of running some form of JVM that and implementation of the OSGi framework can run on, whether it be a scaled down version of Java for embedded processors to the full Java Runtime Environment that one would see on a PC. The JVM is the runtime environment on top of which the OSGi service framework runs. The OSGi framework provides the environment that manages the lifecycle of "bundles" of code, or components. One or more components define a feature, such as a primitive driver for a robot. The OSGi framework supports a Services Oriented Architecture through its service registry – providing a way for components running in the OSGi execution environment to look up other services they need is a loosely coupled way.

A Services Oriented Architecture (SOA) is an architecture that is based on services. Services define a contract that any object implementing that service must follow. This allows for a variety of implementations of the same service. This is one of the key benefits of using SOA in the OSGi runtime environment – different implementations of the same service can be used depending on the situation. This creates a loose coupling between components, so, for example, if one needed to switch to using Ethernet communications as opposed to serial communications, the switch would be trivial as long as both communications features implement the same service contract. The ability to remotely deploy new bundles of code into running OSGi framework environments also contributes to ensuring interoperability, so if a new requirement came out for a change to a communications component for example, a new implementation of the communications service could be sent out to the device running the OSGi framework and replace the older service without requiring significant changes in other components due to the loose coupling of services. Figure 4 in section 2.2.2 shows an example of components running on a robot/sensor in an OSGi runtime.

2. MAXIMIZING SOFTWARE REUSE, ENSURING INTEROPERABILITY, AND DYNAMIC RECONFIGURABILITY

The combination of Model Driven Engineering (MDE), use of the OSGi service platform, and use of a Services Oriented Architecture (SOA) is a powerful and very valuable combination for unmanned system and sensor development. MDE provides the software reusability and ease of configuration by allowing users to take pre-existing models that correspond to automatic generation of code and using them to create or modify complete systems. The OSGi + SOA combination provides the reconfigurability and helps ensure future interoperability because of its loose coupling of services, ability for remote deployment of new or updated features, and because it allows multiple implementations of the same service to remain invisible to the features requiring that service.

2.1 Model Driven Engineering Approach

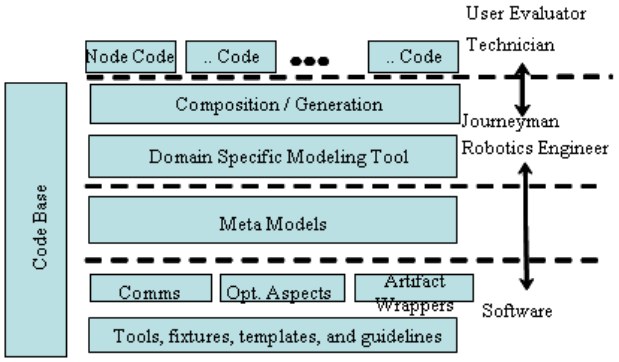

Figure 3: Diagram of Model Drive Engineering Process

Figure 3 shows the MDE approach we take visually as a block diagram of high level abstractions. The placement of the abstractions indicates a progression from a flexible sub-architecture in the realm of the software engineer at the bottom, to a generic sub-architecture in the realm of the robotic/unattended sensor engineer in the middle, and finally to the user/evaluator realm at the top of the figure. The MDE approach is critical to reuse of software and efficient, domain level design of software systems.

2.1.1 Code Base

The code base block on the left of figure 3 represents some abstract storage mechanism, which could be a database, files, or some other mechanism. Code (in this case services and their implementations that will run in the OSGi environment) is stored in this storage mechanism and is used in the MDE approach.

2.1.2 Foundation

The block at the very bottom of figure 3 represents the foundation for the MDE design approach – it includes tools, guidelines, requirements, standards, etc. This foundation is used as a base for higher levels in the MDE process.

2.1.3 Components

Above the foundation in figure 3, but still in the realm of the software engineer lie a set of blocks that represent reusable components. These components are stored in the code base when completed. Systems are composed using these components. Components can be added and removed as time goes on, but new Meta Models must be created to take advantage of the new components. These components can be implementations of one or more services.

2.1.4 Meta Model

The third block up from the bottom of figure 3 represents the Meta Models. This is the spine of the MDE architecture. Meta Models are created by software engineers with knowledge of the domain or software engineers collaborating with domain experts (i.e. a software engineer collaborating with a robotic systems expert). The Meta Model encapsulates high level information about the system and defines component relationships and constraints.

2.1.5 Domain Model

The fourth block up in figure 3 is the domain specific modeling tool. This tool is generated from the Meta Model – it is a workspace from which concrete models of the system under construction may be instantiated, and is the level at which the domain expert operates.

2.1.6 Composition/Generation

The composition/generation block is where everything is put together – components in the domain model are assembled together by a domain expert to create the software necessary for the system to establish some task. This leads to the final block at the top of figure 3, the node code, which is the actual code needed to run the software for the system.

2.2 OSGi and a Service Oriented Architecture (SOA) for Ensuring Interoperability and Reconfigurability

Using the OSGi framework with a Services Oriented Architecture (SOA) is the second piece of the puzzle of ensuring software reuse, interoperability, and reconfigurability. The OSGi + SOA combination is the key factor on top of the MDE approach to making unmanned systems and sensors easy and quick to reconfigure and update, and efficient to add new features to, including upgrades that support interoperability with new standards, communications, and hardware.

2.2.1 OSGi – Managing the Lifecycle of Components

Components created using the MDE design approach are eventually deployed as services to an OSGi runtime environment running on a target unmanned system, sensor, or other device. These OSGi compliant components can be deployed from some data/code storage location either from the machine running the OSGi environment itself, or from some remote location over a network or other communications link. The OSGi runtime manages components that can be dynamically installed, started, stopped, updated and uninstalled. This means that as new or updated features are created, they can be deployed to the runtime environment on a robot or unattended sensor with little or no impact to proper function of the robot or unattended sensor. Using OSGi along with the library of services that can be created through the MDE design approach leads to the capability to rapidly develop and reconfigure components for remote deployment to an unmanned system or sensor. Along with a loosely coupled SOA, this leads to the ability to ensure that new features and payloads can rapidly be added to robots or sensor networks, and to make sure that code can quickly be updated to comply with changing standards and requirements.

2.2.2 Using a Services Oriented Architecture with OSGi to Ensure Interoperability and Dynamic Reconfigurability

Generating components that implement services in the MDE design approach and using those service provides (components) in the OSGi runtime on a robot, unattended sensor, or other device allows for a very easy way to make sure that as standards and requirements change, that a robot or unattended sensor can keep up with those requirements. Figure 4 illustrates a simple Services Oriented Architecture based environment running on a robot using an OSGi framework.

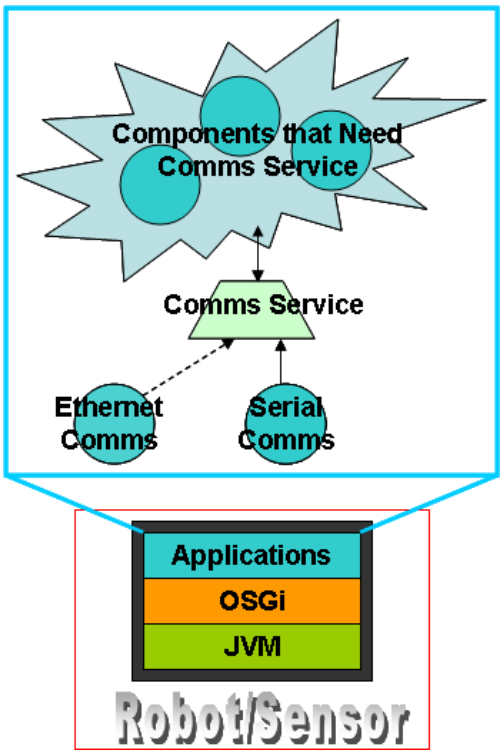

Figure 4: OSGi Setup on a Robot/Sensor using a Services Oriented Architecture

In figure 4, there are two implementations of a Communications (Comms) Service – these are components running in the OSGi runtime environment. For each of these components, there is configuration information indicating that they are registering that they export a Comms Service. This is essentially telling the Service Registry in the OSGi runtime environment that if any other component requires a Comms Service, both of these two implementations can provide that service. In the case of figure 4, there are several components that need a Comms Service to talk to the outside world, perhaps an OCU. Let us call one of these components requiring Comms Service the Robot Commander. The Robot Commander doesn't care how it communicates – all it knows is that it needs some implementation of a Comms Service. The actual way that one of the services is picked is dependent on the OSGi environment and user preference, but it could be based on some configuration data, the most current version of a service could be used, or a random service could be grabbed if no negative effects were expected. It is because of the ability to update or deploy new components into the OSGi runtime, along with the loose coupling of the Robot Commander to the Comms Service implementation, that making sure systems stay interoperable is relatively simple. In this case, if we were required to change the way we do communications with the Robot Commander due to requirements for talking to new systems, all we would have to do is create a new component that complies with the new requirement and deploy it to the OSGi runtime environment on the robot. The Robot Commander could then use that new Comms implementation without requiring any changes to itself since the new communicator implements and exposes the Comms Service that the Robot Commander needs.

2.3 MDE + OSGi + SOA – Wrapping them All Together

Generalizing the example of figure 4, it is easy to see how combining the use of MDE, OSGi, and SOA leads to unmanned systems and sensors that can be rapidly developed, easily reconfigured, and kept up to date with new interoperability standards and features. Using the MDE design approach allows significant reuse of code and reduction of effort by storing components created in a code base. If these components are designed as OSGi compliant service implementations, then they can be remotely deployed to an OSGi runtime environment on a robot or sensor device and can register the services that they provide for use by other components in the environment. By having components only depend on services and not implementations, updating capabilities, including adding new features or making changes to ensure interoperability, can be done by simply deploying new components that expose the required services for use by existing components. This section is a somewhat high level description of the key concepts of software reuse, ensuring interoperability, and reconfigurability – section 3 uses a detailed concrete example to better show how the MDE + OSGi + SOA works.

3. EXAMPLE: UNMANNED SYSTEM DESIGN USING MDE, OSGI, AND SOA

In this section, we describe a situation in which we have a simple robot that we need to design software for. Initially, the robot has the following design: a Driver for steering and driving around and a Communicator that receives drive commands. The design also includes a Bump Sensor which we initially do not have – when the Bump Sensor is present, it can be used to prevent drive commands from running the robot into a non-passable boundary. The Bump Sensor utilizes a 360 degree Sonar Sensor to determine when it is about to "bump". There is also an OCU that sends the drive commands to the robot's Driver through the Communicator.

We first describe how we use the MDE design approach to define a Meta Model for the robot, including its Driver and Communicator components. After that, we add a new capability, the bump sensor, and show how utilizing SOA and OSGi it can rapidly and seamlessly be integrated into the robot system. After that, a new requirement comes in to make the robot JAUS compliant and able to receive JAUS drive commands from the OCU – this shows how using OSGi and SOA, along with reusing components from the original MDE design process, we can rapidly reconfigure the robot to make sure it is interoperable with JAUS OCUs and other JAUS robots. Finally, we summarize the entire process and discuss its extension to use on a real robot.

3.1 MDE Approach to Create Robot System

We step through figure 3 to lay out step by step the MDE design approach for creating the robot system that we desire.

3.1.1 Code Base

In this case, the code base will consist of java components written to implement service contracts. The services that will be implemented are the Communicator Service, the Driver Service, Sonar Service, and a Bump Sensor Service. Each implementation of these services will be packaged as a jar for deployment into the OSGi runtime environment.

3.1.2 Foundation

We will use the Generic Modeling Environment software (GME, 2006), an open source, visual, and configurable environment for creating Domain Specific Modeling languages, to develop our Meta Model. The Eclipse IDE will be used for working with Java code and for using the Equinox OSGi (Eclipse, 2006) runtime for

deploying bundles to – this is also chosen because it is widely used and open source. The domain model will be represented by XML once completed. For now, our robot will be simulated in a Java-based simulation environment, but later sub-sections will explain how the combination of OSGi + SOA would allow quick transition into a real robot.

3.1.3 Components

For the initial design of the robot, we need the following components: Driver implementing Driver Service and Simple Communicator implementing Communicator Service. Initially, we will not have a bump sensor component, but will still need to define the Bump Sensor Service for use by the Driver – in this case we use a Fake Bump Sensor that always indicates nothing is being bumped, effectively letting the Driver function without a filter checking for collisions. The Sonar Sensor will implement a Sonar Service, and will be used by the bump sensor. The Driver will move the robot around the simulated environment, and the communicator will receive messages from an OCU containing drive commands as simple text messages like "message=drive&xeffort=[XEFFORT]&yeffort=[YEFFORT]&rotation=[ROTATION]". *Xeffort* is the effort as a percentage total possible speed that the robot should move in the x direction, and *yeffort* is the percentage of the total possible speed of the robot in the y direction. *Rotation* is the rotational speed of the robot in rad/s. Figure 5 in 3.1.5 illustrates graphically the setup of the robot system.

3.1.4 Meta Model

We use the Meta Model in figure 1 for this example. We will have a number of sensors on our robot, and it will communicate with the OCU. These objects and their relationships to each other are defined in the Meta Model in figure 1.

3.1.5 Domain Model

The Domain Model we use descends from the Meta Model of figure 1. We have defined a number of components based on the Meta Model of figure 1. Figure 5 shows the Domain Model for our simulated robot system. We will have a robot ("ODIS") containing the Driver and Communicator connected to an OCU. It will make use of a Bump Sensor ("BumpControl") to act as a filter for the Driver – when implemented, the Bump Sensor will use a simulated 360 degree sonar to determine when something is "bumped". As can be seen, this Domain Model is a simple block diagram that can easily be assembled with someone with knowledge of robotics but without any software engineering expertise. The blocks (and masked sub-blocks) shown in figure 5 are also placed into the code base as available components for deployment.

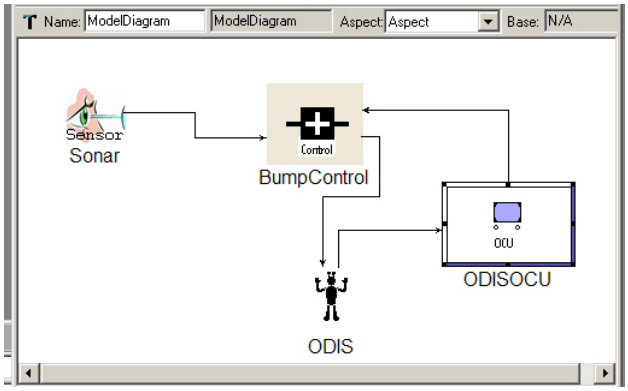

Figure 5: Robot Bump Control Domain Model

To create this Domain Model, we, in the role of the Domain Engineer, have connected the components with arrows representing information paths and overrode the default naming attributes.

3.1.6 Composition/Generation

The output of the Domain Model we have created is a specification for an application based on constraints and relationships from the Meta Model and manipulation of the model by us in the role as the Domain Engineer. A set of components forming our robot application is generated from the specification that is outputted from the domain model, along with the information from the Meta Model. These components are now available in the code base, and are ready for deployment.

3.2 Deployment of Components to OSGi Runtime Environment on Robot

We now have generated the code needed to run on the robot through the MDE process. What needs to be done now is to deploy the components that have been developed onto the robot. For now, since we are simulating the robot, the components will be deployed to the same platform as the code base storage. We use the Equinox framework from Eclipse (Eclipse, 2006) as our OSGi implementation.

For code base storage, we are currently just storing a set of jars representing the various components in a folder on the file system. At startup, the equinox platform is only running some core platform jars – we must add our components to the runtime so that they may be started and used. We simply install each component jar into the runtime environment – at this point they are not active, but they do contain information about what services they provide, what services they need, and other configuration and deployment information. We can now tell the Equinox runtime to start the individual components – using configuration files packaged with the component jars, components that depend on certain services (like how the Driver requires the Communicator Service) will look for that service in the runtime environment and start

it if found (in our case, since we have installed all components, it is found). Now that all the components have been started, we have a functional (simulated) robot that can drive around the simulation environment using its Driver, and can receive commands from an OCU or other source using the Simple Communicator. In this case, the OCU is just a simple java app, but it could easily also have its own OSGi runtime environment and be using the same Communicator Service and components as the robot. Figure 6 shows how components running in the robot's OSGi runtime depend on each other – dashed lines indicate components that are not in the initial deployment, but will be put on the robot in place of another component for various reasons in Case 1 and Case 2 to follow.

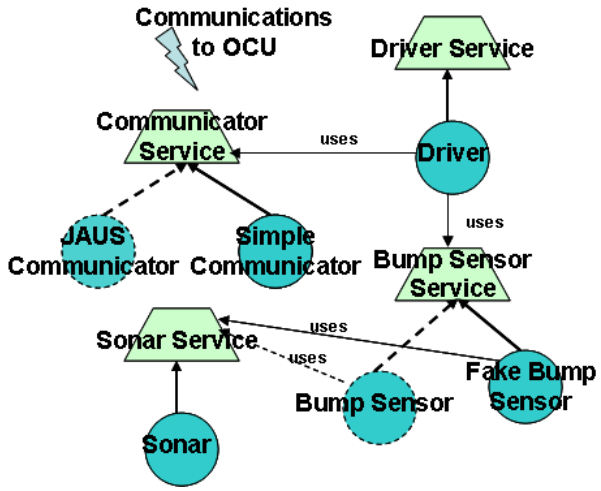

Figure 6: Diagram of the Components Running on the Robot and the Service they Implement/Use

3.3 Case 1: Addition of a Real Bump Sensor

As we run the simulation environment with the OCU providing commands to the simulated robot, we notice that it is possible to drive the robot into the boundaries (walls) of the simulation. Clearly, running into walls in the real world is not desired, and we'd like to find a way to prevent this in the simulation that can be applied to a real robot. We now add a real bump sensor to the robot to replace the simple fake one. The Bump Sensor serves as a filter for the Driver component – if the bump sensor indicates the robot is running into something, its status will change and the Driver will use that status to override the command from the OCU and stop the robot before it makes impact with the wall.

Using the MDE process and OSGi with a Service Oriented Architecture, it is a simple and elegant process to implement this real bump sensor. First, the domain model must be updated to use a real Bump Sensor component in place of the generic fake one. This can be done at the domain specific level by the robot engineer assuming a real Bump Sensor component already exists in the code base. The Fake Bump Sensor running in the OSGi runtime on the robot is next stopped and/or uninstalled remotely or locally, and the new Bump Sensor is sent to the robot either from a local or remote code storage source. Since both the Fake Bump Sensor and the real Bump Sensor implement the same Bump Sensor Service, the transition will be seamless to the Driver component that is using the Bump Sensor Service – it will now be able to use the new Bump Sensor and get accurate information on when it is bumping into a wall or other obstacle.

3.4 Case 2: JAUS Requirement Added

Up until now, we have been using a Simple Communicator that uses basic text based messages. This simulates what might be a proprietary or legacy communications method. Using the architecture and methods we have explained, updating this to meet a new JAUS requirement is very straightforward. JAUS, the Joint Architecture for Unmanned Systems (soon to be an SAE standard) is a message-passing architecture for communications among unmanned systems (JAUS Working Group, 2006). All we must do is replace the Simple Communicator with a new JAUS Communicator just as we did for the Bump Sensor. Again, since the JAUS Communicator implements the Communicator Service just like the Simple Communicator did, the change will be transparent to all components that depend on having a Communicator Service and the robot will now be interoperable with JAUS. Another thing to note is that if we wanted to have an even looser coupling in our system, we could have each component depend only on a single messaging service – each component could bind to this service and send messages through it, allowing the messaging service to route messages as needed and send messages in the proper format (i.e. custom format vs. JAUS standard).

3.5 Summary and Extension to a Real Robot

In this section, we have shown how using a combination of Model Driven Engineering with a runtime environment of OSGi and a Services Oriented Architecture, that applications for unmanned systems and sensors can rapidly be developed and deployed, both locally and remotely. However, we have used a simulated robot – clearly, we want to apply this to real robots. This is where we see the true power of using the Service Oriented Architecture. As already seen through addition of the real Bump Sensor and JAUS Communicator, it is a simple process to add new implementation of services to the runtime on a robot, unattended sensor, or other device that is running an OSGi runtime. The fact that we are using a Services Oriented Architecture where components depend on services and not specific implementations allows us to create very flexible and upgradeable systems.

If we know that there are sensors or payloads that we will want to use or support, but do not have available at the time, we can use a simulated version of that device until a real one is available. This allows for an environment of simulated and real devices in which testing can be done even if not all the necessary hardware is available at the time. Adding new hardware is also simple, as the communications, transport, and device code needed can be remotely sent to the OSGi runtime when needed. Updates can be as simple as adding the new component and binding it to a messaging service that all components talk to each other through – i.e. a new JAUS component could be added to a robot and "hooked up" to a JAUS messaging service that routes JAUS messages to their proper destinations.

4. RELATED WORK

4.1 Chrysler AG

Czarnecki, Bednasch, Unger and Eisenecker report on their experience at Chrysler AG for automotive and satellite applications (Czarnecki et al, 2002). They describe their experience with Model Driven Design and Feature Modeling tool support with the GME tool.

The feature model has a root or concept node and child nodes. The child nodes or sets of child nodes are mandatory, optional, alternative or "or" features. The nodes are combined in various ways to produce an instance of a concept. For example, a car (concept) can have a manual, automatic or CV transmission, but only one transmission. A car may also have a fossil fuel motor, and electric motor or both.

In the referenced work, they present a UML Meta Model for feature modeling notation using GME. They also show a derived domain specific model, also using GME.

4.2 Embedded System Control Language

Additional work at Vanderbilt University uses the GME tool, along with Mathworks Simulink and Stateflow tools to create the Embedded Control Systems Language (ESQL) to support development of distributed embedded automotive application (GME, 2006). ESQL imports the Simulink/Stateflow models into the GME environment. ESQL is a graphical modeling language for with a suite of sublanguages. Sublanguages are provided to support functional modeling, component modeling, hardware topology modeling and deployment mapping.

The ECSL also has a code generation component. The generated artifacts can synthesize the entire application behavior code, or external application behavior code can be linked in.

4.3 TARDEC Robotics Mobility Lab (TRML) Innovative Laboratory Internal Research (ILIR)

The TRML, in cooperation with the Naval Postgraduate School (NPS) is conducting research into software reuse via composable components and graphical configuration languages. This work is the basis for the MDE portions of this paper.

The major difference between this effort and the two cited above is the target users and target life cycle time. The above tools are being designed for software engineering production departments. The TRML effort is targeting Domain engineers, early in the system lifecycle.

Prototype systems are typically created in response to evolving requirements or to evaluate emerging technologies, such as sensors or mission packages. In some cases, rapid prototypes and/or simulations are necessary to evaluate procurement proposals.

The main goals of the TRML efforts are to capture and encapsulate software engineering expertise into a tool. The tools we are creating extend the software engineers knowledge in a constrained environment operated by Domain engineers to rapidly create prototype systems.

4.4 National Automotive Center's Pervasive Computing Lab.

The OSGi and Services Oriented Architecture approaches in this paper are based on similar ongoing work at the National Automotive Center's Pervasive Computing Lab. The work has involved integration of various sensors and other applications on to both vehicle and hand held (tablet PC) platforms under a project called Cyrano. Using the OSGi and SOA concepts described in this paper, we have been able to rapidly develop applications for vehicle diagnostics using data from the vehicle bus, situational awareness, chemical and radiation sensors, GPS, and a wide variety of other devices and communications protocols. Using a Services Oriented Architecture with OSGi has allowed us to very quickly develop applications based on customer feedback and requirements, in as little as a week or two. The SOA architecture also allows us to utilize simulated and fake devices and sensors interchangeably with the real things, as some of the devices are often unavailable, expensive, or provided on a time-limited basis.

5. FUTURE WORK

Future work with MDE includes implementation of the design environment for prototyping a series of robotic systems. Beginning with simple models, robot simulations and very coarse grained components, the

simple models presented earlier will be realized. Continuing, the Meta Models will be refined to include lower level component composition. A set of robotic artifacts (platforms, controls, OCU's etc.) will have their interfaces wrapped to conform to the JAUS standard. A collection of instrumentation components will be created, as well as several different communications components; UDP/IP and serial to begin with. As we grow more confident with the Meta Models and domain specific models, additional artifacts such as mission packages and manipulators will be included both in simulation and physically.

Future work involving the OSGi and SOA concepts of this paper will focus mainly on deployment of applications of the concepts onto actual robot platforms. This may involve integration with the related work that has been done with sensor integration on vehicle and tablet PC platforms. A robot could easily be integrated as another application to augment the sensor information and other applications of project Cyrano – the OCU could in this case be the preexisting user interface on a tablet PC or any other computing device supporting an OSGi runtime. Farther in the future, work would include use of an OSGi runtime on a fully functional, useable robot (as opposed to just a research platform) with the goal to create a robot based on a flexible, open architecture that is easy and cost effective to update and reconfigure.

6. CONCLUSIONS

In this paper, we have detailed and demonstrated through example three key concepts for development of unmanned systems and sensors – Model Driven Engineering, OSGi, and a Services Oriented Architecture. We have shown that, by using a Model Driven Engineering approach, components can be developed by software engineers and then combined into a system by someone with domain level expertise (i.e. a robotics engineer) but without a lot of software engineering experience. We have shown that by basing these components on a Services Oriented Architecture, we can create extremely flexible and reconfigurable designs that allow for a mix of simulated and real hardware, which is extremely powerful during testing and development. Finally, we have shown how using OSGi along with a Services Oriented Architecture allows us to rapidly deploy new features to robotic systems and unattended sensors in order to help ensure they remain interoperable with other systems, and rapidly update their capabilities when needed. All of these concepts combine to allow us to rapidly develop and update unmanned systems and sensors as soon as requirements and feedback come in.

REFERENCES

Czarnecki, K., Bednasch, T., Unger, P., and Eisenecker, U., 2002: Generative Programming for Embedded Software: An Industrial Experience Report, *Proceedings ACM SIGPLAN/SIGSOFT Conference*, GPCE, Pittsburgh, PA, October 2002.

Eclipse, 2006: Equinox, http://www.eclipse.org/equinox

Generic Modeling Environment, 2006: http://www.isis.vanderbilt.edu/projects/gme

JAUS, 2006: JAUS, http://www.jauswg.org

OSGi Alliance, 2006: Welcome to the OSGi Alliance, http://www.osgi.org

Schmidt, D., 2006: Model Driven Engineering, *IEEE Computer, February 2006*, 25-31